A Programming and Execution Environment for Distributed Multi Agent Systems *

John R. Graham, Daniel McHugh, Foster McGeary,
M. Victoria Windley, David Cleaver, Keith S. Decker

University of Delaware
Newark, DE, 19716, USA
{graham,mchugh,mcgeary,windley,cleaver,decker}@cis.udel.edu

ABSTRACT

Before the powerful agent programming paradigm can be adopted in commercial or industrial settings, a complete environment, similar to that for other programming languages must be developed. This includes, editors, libraries, and an environment for the completion of agent tasks. The DECAF[8] Agent architecture is a general purpose agent development platform that was designed specifically to support concurrency, distributed operations, support for high level programming paradigms and high throughput. The architecture has been designed with built-in scalability which adapts itself to multiple processor architecture and highly distributed multi-agent systems. DECAF supports research efforts in planning and scheduling with modular design. The architecture also supports application development and has current developments in social modeling, middle agents, information extraction and proxy operations. DECAF also supports the next step in the progression of the programming paradigm by allowing "flexible" and "structured persistent" actions [7]. This paper is a case study of the development of the DECAF architecture, tools that have been developed concurrently to support programming and testing, and some of the more significant applications designed using DECAF.

1. INTRODUCTION

Providing an environment for the execution of a software agent is very similar to building an operating system for the execution of general purpose applications. In the same fashion that an operating system provides a set of services for the execution of a user request, an agent framework provides a similar set of services for the execution of agent actions. Such services include the ability to communicate with other agents, maintaining the current state of an executing agent, and selecting an execution path from a set of possible execution paths.

Agent systems are composed of a collection of autonomous units that have local information and local capabilities. In multi agent systems, local information and goals are normally insufficient to achieve larger goals independently. To support the achievement of larger, non-local goals, agents must communicate and exchange information with other agents. This scenario of solving problems imposes design constraints which an agent architecture must support in order to be effective. **Communication.** Agents are distributed across networks and need to communicate. **Concurrency.** Concurrent activities are essential for an architecture in order to improve the availability of its services. Working on task solutions at the same time as processing incoming messages, for example. **Language Support.** Agent programming includes all of the usual programming paradigms as well the extensions that make agent programming unique such as flexible actions. **Testability.** Repeatability is essential for designing solutions to complex tasks.

There are two major design features of DECAF that support these design constraints. First, DECAF consists of a set of well defined control modules (initialization, dispatching, planning, scheduling, execution, and coordination) that work in concert to control an agent's life cycle. Each of the modules is implemented as a separate thread in Java. Secondly, there is one core task structure representation that is shared between all of the control modules. The task structure can be annotated and expanded as needed with details that may be understood by only one or two modules, but there is still one core representation.

In addition, a separate goal of the architecture is to develop a modular platform suitable for our research activities. DECAF distinguishes itself from many other agent toolkits by shifting the focus away from the underlying components of agent building such as socket creation, message formatting, and the details of agent communication to allow agent developers to focus on the logic of the task or in the case of research, allowing focus on one particular aspect of the architecture.

In support of these goals a set of tools has evolved to assist programmers and researchers; Component libraries, GUI programming, Agent Management Agent (AMA) and middleware (ANS (agent name server), Matchmaker and Broker). The remainder of this paper will briefly discuss each of the control modules and the support tools. Lastly an overview of the research projects and applications that have been developed using DECAF.

2. THE DECAF ARCHITECTURE

DECAF (Distributed, Environment-Centered Agent Framework) is a toolkit which allows a well-defined software engineering approach

to building multi-agent systems. The toolkit provides a stable platform to design, rapidly develop, and execute intelligent agents to achieve solutions in complex software systems. DECAF provides the necessary architectural services of a large-grained intelligent agent [5, 14]: communication, planning, scheduling, execution monitoring, coordination, and eventually learning and self-diagnosis [9]. This is essentially the internal "operating system" of a software agent, to which application programmers have strictly limited access. The overall internal architecture of DECAF is shown in Figure 1.

2.1 Agent Initialization

The execution modules control the flow of a task through its life time. After initialization, each module runs continuously and concurrently in its own Java thread. When an agent is started, the *Agent Initialization* module will run. The agent initialization module will read the plan file as described above. Each task reduction specified in the plan file will be added to the *Task Templates Hash table* (plan library) along with the tree structure that is used to specify actions that accomplish that objective.

2.2 Dispatcher

Agent initialization is done once and then control is passed to the Dispatcher which waits for an incoming KQML (or FIPA) message. These messages will then be placed on the *Incoming Message Queue*. An incoming message contains a KQML *performative* and its associated information. An incoming message can result in one of two actions by the dispatcher. First the message is attempting to communicate as part of an ongoing conversation. The Dispatcher makes this distinction mostly by recognizing the KQML :in-reply-to field designator, which indicates the message is part of an existing conversation. In this case the dispatcher will find the corresponding action in the *Pending Action Queue* and set up the tasks to continue the agent action.

Second, a message may indicate that it is part of a new conversation. This is the case whenever the message does not use the :in-reply-to field. If so a new *objective* is created (equivalent to the BDI "desires" concept[12]) and placed on the *Objectives Queue* for the Planner. An agent typically has many active objectives, not all of which may be achievable.

2.3 Planner

The Planner monitors the Objectives Queue and matches new goals to an existing task template as stored in the Plan Library. A copy of the instantiated plan, in the form of an HTN corresponding to that goal, is placed in the *Task Queue* area, along with a unique identifier and any provisions that were passed to the agent via the incoming message. If a subsequent message comes in requesting the same goal be accomplished, then another instantiation of the same plan template will be placed in the task networks with a new unique identifier. The Task Queue at any given moment will contain the instantiated plans/task structures (including all actions and subgoals) that should be completed in response to an incoming request.

2.4 Scheduler

The *Scheduler* waits until the Task Queue is non-empty. The purpose of the Scheduler is to determine which actions *can* be executed now, which *should* be executed now, and in what order they should be executed. This determination is currently based on whether all of the provisions for a particular module are available. Some provisions come from the incoming message and some provisions come as a result of other actions being completed. This means the Task Queue Structures are checked any time a provision becomes available to see which actions can be executed now.

A major research effort is underway to add reasoning ability to the scheduling module. This effort involves annotating the task structure with performance and scheduling information to allow the scheduler to select an "optimal" path for task completion. Optimal in this case may mean some definition of quality or deadline and real-time goals.

2.5 Executor

The *Executor* is set into operation when the Agenda Queue is non-empty. Once an action is placed on the queue the Executor immediately places the task into execution. One of two things can occur at this point: The action can complete normally (Note that "normal" completion may be returning an error or any other outcome) and the result is placed on the *Action Result Queue*. The framework waits for results and then distributes the result to downstream actions that may be waiting in the Task Queue. Once this is accomplished the Executor examines the Agenda queue to see if there is further work to be done. The Executor module will start each task in its own separate thread improving throughput and assisting the achievement of the real-time deadlines. Alternatively, an action may fail and not return, in which case the framework will indicate failure of the task to the requester.

3. ARCHITECTURE DEVELOPMENT

Currently, DECAF is being used as a research platform for some classical AI problems in scheduling and planning. The following sections briefly describe these research efforts.

3.1 Scheduling

DECAF supports the idea of "soft" real time execution of tasks. To achieve this, the concept of execution profiles and a characterization of agent execution that will lead to optimal or near optimal scheduling of agent execution has been developed. This work is leveraged from the Design to Criteria (DTC) work at the University of Massachusetts [15]. Using the agent execution profile and characterization, Currently, DECAF can be run with a simple non-reasoning scheduler or with the DTC scheduler. Under development is DRU (Dynamic Realtime Update), a scheduler that is faster than DTC and improves reliability by taking advantage of the Java Virtual Machine (JVM) to run redundant efforts to achieve deadlines in the event of failure of the primary solution.

3.2 GPGP

Generalized Partial Global Planning (GPGP) is a task structure centered approach to coordination [3]. The basic idea is that each agent constructs its local view of the structure and relationships of its intended tasks. This view is then augmented by information from other tasks and the local view will change dynamically over time. In particular, commitments are exchanged that result in new scheduling constraints. The result is a more coordinated behavior for all agents in the community.

3.3 Planning

The focus of planning in our system is on explicating the basic information flow relationships between tasks and other relationships that affect control flow decisions. Most control relationships are
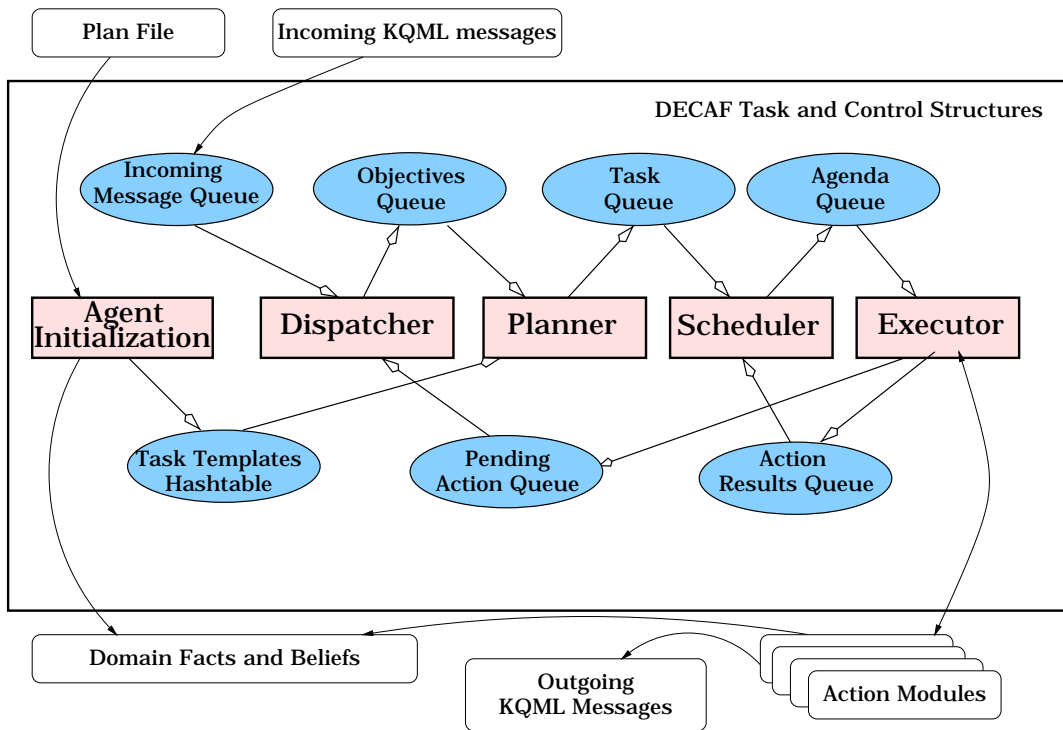
Figure 1: DECAF Architecture Overview

derivative of these more basic relationships. Final action selection, sequencing and timing are left up to the agent's local scheduler. Thus the planning process takes as input the agent's current set of goals and set of task structures and produces as output a new set of current task structures. The important constraint on the planning module is to guarantee at least one task for each goal until the goal is accomplished or until the goal is believed to be unachievable.

Special features of the planner include the ability to plan for preconditions and plan to achieve abstract predicate goals (instead of decomposition by task names). The planner also designs plans to allow runtime choices between branches to be made by an intelligent scheduler, based on user preferences that can change between plan time and runtime. This feature provides real time flexibility, since the scheduler can react to a dynamic environment by exploiting choice within a plan, rather than forcing the planner to do costly replanning.

4. AGENT DEVELOPMENT TOOLS

4.1 Plan Editor

The control or programming of DECAF agents is provided via an ASCII *Plan File* written in the DECAF programming language. The plan file is created using a GUI interface called the *Plan-Editor*. In the Plan-Editor, executable actions are treated as basic building blocks which can be chained together to achieve a larger more complex goal in the style of an HTN (hierarchical task network). This provides a software component-style programming interface with desirable properties such as component reuse (eventually, automated via the planner) and some design-time error-checking. The chaining of activities can involve traditional looping and if-then-else constructs. This part of DECAF is an extension of the RETSINA and TÆMS task structure frameworks [16, 2].

The DECAF Plan-Editor attaches to each action a performance profile which is then used and updated internally by DECAF to provide real-time local scheduling services. The reuse of common agent behaviors is thus increased because the execution of these behaviors does not depend only on the specific construction of the task network but also on the dynamic environment in which the agent is operating.

For example, a particular agent may be "persistent", or "flexible" [17] meaning the agent will attempt to achieve an objective, possibly via several approaches, until a result is achieved. This construction also allows for a certain level of non-determinism in the use of the agent action building blocks. Figure 2 shows a PE session.

The PE facilitates code generation in multiple languages through the use of a common data structure, the *PEComponentData*. A PEComponentData is created to represent each object in a Plan File (tasks, actions and non-local actions). This representation is independent of any particular language and is easily used to generate any output language. Currently the PE can generate compilable Java code, the DECAF Language, and TÆMS. This process is easily be replicated for generation of any language by creating a new Java method to translate the PEComponentData into another source language.

4.2 Agent Construction

One of the major goals of building the DECAF framework is to enable very rapid development of agent actions and task structures. This is accomplished by removing the agent interaction details from the programmers hands. The developer does not need to write any communications code, does not have to worry about multiple invocations of the same agent, does not have to parse and schedule any incoming messages, and does not have to learn any
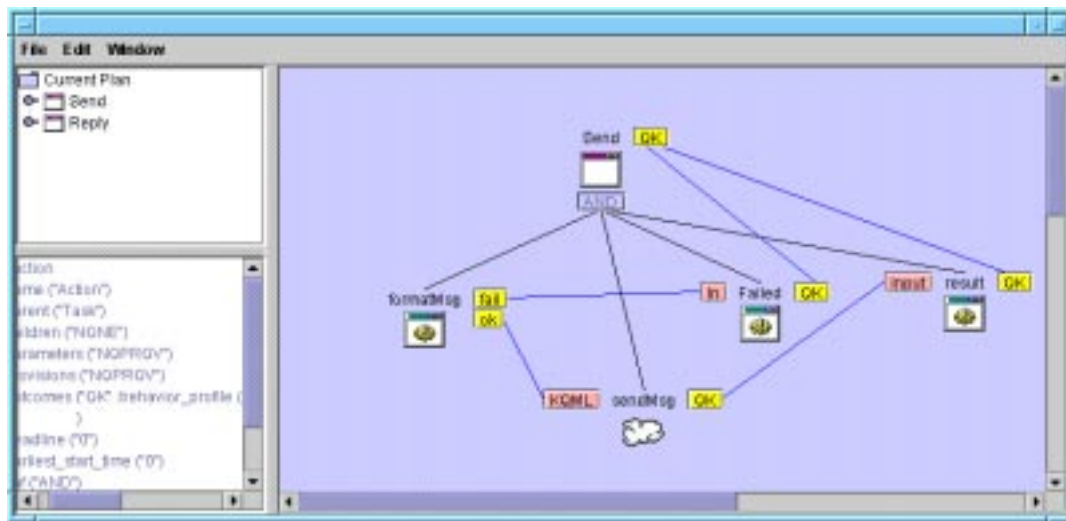
Figure 2: Sample PE Session

Application Programmer Interface (API) in order to write and program an agent under the DECAF architecture. Note that since the Plan File incorporates all of the data flow of an agent, the programmer does not have to write code for data flow between actions either.

The plan file represents the agent programming and each leaf node of the program represents a program that the user must write. The DECAF language supports all of the usual programming constructs such as selection and looping, but it also supports the idea of "structured persistent actions". One feature of an agent oriented approach to problem solving is the ability to describe in broad terms the method for achieving a goal. The programmer does not have to build from scratch an explicit solution, rather they can build an encapsulation that continuously tries for success without accounting for all possible conditions. For example, to look up a quote for a stock may require may queries to many price databases. The programmer does not have be concerned with details of which database is used or what remote format the data is in, only that eventually the price will be returned or that the price is not available.

The PE session in Figure 2 demonstrates some important programming concepts built into DECAF. In this case the task is named "Send". Generically this task will format a KQML message, send it to the designated recipient and await the response. There are three actions the programmer must write and all the rest is handled by DECAF. If the "formatMsg" task fails, the failure is reported to the "Failed" action which does the error processing and reports that the task is complete. Otherwise, the message is sent to the cloud construct. In DECAF the cloud represents a non-local action or task. Internally the cloud takes cares of all communication, timeouts and retries for message delivery and ultimately delivery of the response to the downstream action. This is true if there is one message or a stream of messages to be sent or received. All that remains for the programmer is to process the replies.

4.3 Test Generation Tool

To properly test the DECAF architecture, it is necessary to generate random plan structures. However, simply generating random structures is not enough. The test generation program for DECAF

allows random plan files to be created with certain features, that allow testing of different aspects of DECAF. These features include the average depth of the tree, the average breadth, and the amount of enablements within a tree level. With only a few minor parameter changes, many different types of plan files can be quickly generated and used.

5.   MIDDLEWARE

Middle agents support the flow of information in a MAS(Multi-Agent System) community. They do not contribute directly to the solution of the goal. For example, a middle agent that lists all of the airlines traveling from New York to Chicago does not find you a ticket for the trip. However, in order to get such a ticket, you need the list of airlines. You could of course program such functionality into your ticket finding program but it is easier to have such a list available as a middle agent.

5.1   Agent Name Server

The current DECAF Agent Name Server is based on a version in use at Carnegie-Mellon University. This is a stand alone program with a fixed API which does the registration of agents. The new DECAF ANS under development will be written as a DECAF agent. This will allow interaction with the ANS through KQML messages. It also allows new functionality to be easily added via new task structures to a DECAF plan. To avoid excessive message traffic and to maintain directories, the ANS agent will have a known port and listen to simple socket connections from other agents. Simple activities with the ANS, such as registering, unregistering and looking up other agents can be handled by these simple socket connections. These simple protocols and other more complex ones can be handled through the normal message port of the ANS. This design will lead to more complex behaviors by the ANS. Agent Name Server Agents could register with each other, and a protocol could be developed similar to DNS for finding an agent's location given it's name as well as for increased security.

5.2   Matchmaker and Broker Agents

Two middle agents that have been developed using DECAF are *Matchmaker* and *Broker*. The Matchmaker agent serves as a yellow pages tool to assist agent in finding other agents in the community

that may provide a service useful for their tasks. An agent will *advertise* its capabilities with the Matchmaker and if those capabilities change or are no longer available, the agent will *unadvertise*. The Matchmaker stores the capabilities in a local database. A requester wishing to ask a query will formulate the query to the Matchmaker and *ask* for a set of matching advertisements. The requester can then and make request directly to the provider. A requester can also *subscribe* to the Matchmaker and be informed when new services or interest are added or removed.

A *Broker* agent advertises summary capabilities built from all of the providers that have registered with one Broker. The Broker in turn advertises with the Matchmaker. For example, one Broker may have all the capabilities to build a house (plumber, electrician, framer, roofer, . . .). The broker can now provide a larger service than any single provider can, and often manage a large group of agents more effectively.

5.3   Information Extraction Agent
The main functions of an information extraction agent (IEA) are [4]: Fulfilling requests from external sources in response to a *one shot query* (e.g. "What is the price of IBM?"). Monitoring external sources for *periodic* information (e.g. "Give me the price of IBM every 30 minutes."). Monitoring sources for patterns, called *information monitoring* requests (e.g. "Notify me if the price of IBM goes below $50.")." These functions can be written in a general way so that the code can be shared for agents in any domain.

Since our agent operates on the Web, the information gathered is from external information sources.The agent uses a set of *wrappers* and the wrapper induction algorithm STALKER [10], to extract relevant information from the web pages. When the information is gathered it is stored in the local IEA "infobase" using Java wrappers on a PARKA [13] database/knowledgbase.

5.4   Proxy Agent
DECAF agent can communicate with any object that uses the KQML or FIPA message construct. However, web browser applets cannot (due to security concerns) communicate directly with any machine except the applet's server. The solution is a *Proxy* agent. The Proxy agent is constructed as a DECAF agent and uses fixed addresses and socket communication to talk to Java applets or any application. Through the Proxy agent, applications outside the DECAF or KQML community have access to MAS Services.

5.5   Agent Management Agent
The Agent Management Agent (AMA) creates a graphical representation of agents which are currently registered with the ANS, as well as the communication between those agents. This allows the user to have a concept of the community in which an agent is operating as well as the capabilities of the agents and the interaction between agents. The AMA frequently queries the ANS to determine which agents are currently registered. These agents are then represented in a GUI. The AMA also queries the Matchmaker to retrieve a profile provided by each agent. This profile contains information about the services provided by an agent. This profile is accessible to the AMA user by double-clicking on the agent's icon. In the future, the AMA will also have the capability of monitoring and displaying communications between these agents. Each agent will send a message to the AMA whenever it communicates with another agent, so that the user may then monitor all activity between agents.

6.   CURRENT DEVELOPMENT
6.1   Modeling with the Virtual Food Court
Virtual Food Court (VFC) is a small artificial economy. VFC models diners, workers, and entrepreneurs. These economic entities are caricatures of the participants in transactions that take place within a simplified shopping mall food court. Although caricatures, the entities exhibit behaviors, chosen from a repertoire of self-interested behaviors, sufficient to allow VFC to contain a labor market, markets for food service equipment, and markets for food products. For example, accepting a contract to perform labor and forming an organization (i.e., offering the labor contract) are reciprocal events. Because both of these are voluntary actions, we believe it necessary to model and explain both sides of the transaction simultaneously. This is what we do in VFC, planning to extend our results to model organizational structures more complicated than a simple employment contract (while still, of course, basing the analysis on the need for there to be reciprocally voluntary contracts). We expect that such models will also have to be expanded to include aspects of governance and perhaps non-economic social forces as we explore the long term control and stability of such structures.

The initial configuration of VFC is shown in figure 3 as a collection of boxes and lines. Lines represent the KQML communications and the boxes are DECAF agents. Arrowheads indicate the direction of the initial message. In the DECAF paradigm, agents need to know of the Matchmaker in order to register their existence with it. Workers and Restaurants need to know of the existence of the Government because they are "required" to report to it.
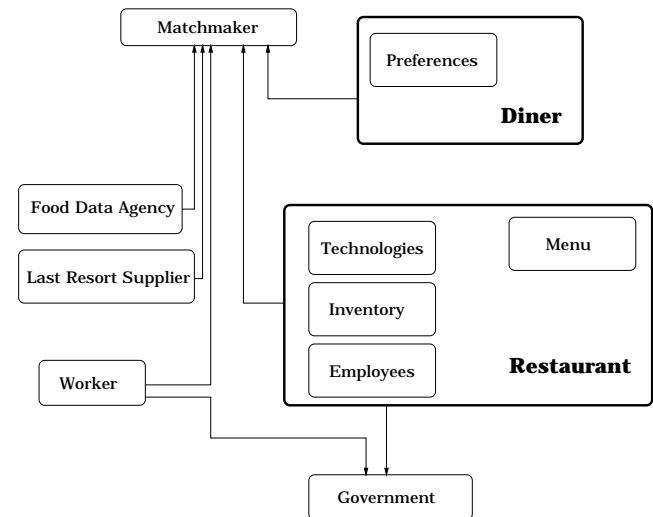

Figure 3: Virtual Food Court Architecture

6.2   GeneAgent
GeneAgent is a bioinformatics-gathering system based on the RETSINA agent organizational concept of Interface Agents that work with humans, Information Extraction Agents that wrap various web resources, and Task Agents that include both middle-agents and domain task analysis agents. GeneAgent interfaces with a biologist via a browser and applets that use the DECAF Proxy Agent to communicate with the rest of the information gathering system. The Information Extraction Agent class has been used to build wrappers for several necessary Internet resources such as the NCBI

BLAST servers that allow searching for protein sequences in GenBank; Protein Motif (sequence pattern) databases such as SwissProt, and local databases of organism-specific genetic sequences. Initial analysis agents provide services such as notification of new BLAST results and automated customized annotation of local genetic sequence information. Figure 4 shows the basic architecture of GeneAgent.
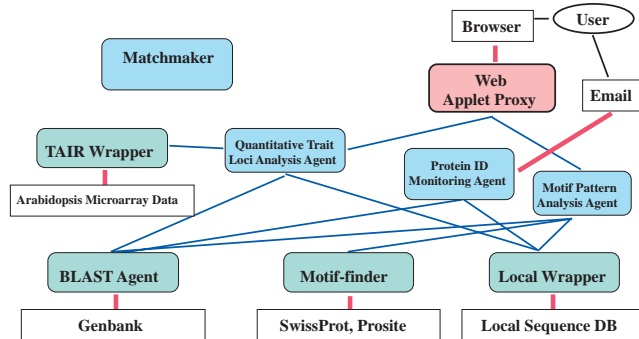

Figure 4: GeneAgent Architecture

7.   CONCLUSIONS AND FUTURE WORK

Currently the most widely used agent development approaches use the "toolkit" concept. Meaning there is an API that the programmer must use to completely build the agent task. Also, there is no convenient representation in a language or GUI that can be written to program the agent. *JATlite* from Stanford University and *Bond* from Purdue University are good examples of this approach. *DESIRE* [1] and *ConCurrent METAtem* [6] are language formalizations but they must be developed by hand and do not allow the coordination mechanisms specified by DECAF and TÆMS. ZEUS [11] is an example of such an extended framework similar to DECAF. ZEUS is a collection of tools (primarily visual) that assist the agent developer in building agent code. As a complete collection of tools it is somewhat more advanced than DECAF. However, ZEUS allows only very simple coordination specification between agents components and does little or no reasoning about agent action scheduling or planning.

DECAF is currently the basis for several AI projects and projects involving organizational development and bioinformatics information gathering. It has also been used as a programming tools for graduate level classes on agent development.

8.   ACKNOWLEDGMENTS

9.   REFERENCES

[1] F. M. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. Desire: Modeling multi-agent systems in a compositional formal framework. *International Journal of Cooperative Information Systems*, 6(1), 1997.

[2] K. S. Decker and V. R. Lesser. Quantitative modeling of complex computational task environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 217–224, Washington, July 1993.

[3] K. S. Decker and V. R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 73–80, San Francisco, June 1995. AAAI Press. Longer version available as UMass CS-TR 94–14.

[4] K. S. Decker, A. Pannu, K. Sycara, and M. Williamson. Designing behaviors for information agents. In *Proceedings of the 1st Intl. Conf. on Autonomous Agents*, pages 404–413, Marina del Rey, Feb. 1997.

[5] K. S. Decker and K. Sycara. Intelligent adaptive information agents. *Journal of Intelligent Information Systems*, 9(3):239–260, 1997.

[6] M. Fisher. Introduction to concurrent metatem. 1996.

[7] L. Gasser. Agent and concurrent objects. *An interview by Jean-Pierre Briot in IEEE Concurrency*, 1998.

[8] J. R. Graham and K. S. Decker. Towards a distributed, environment-centered agent framework. In N. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Proceedings of ATAL-99*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000.

[9] B. Horling, V. Lesser, R. Vincent, A. Bazzan, and P. Xuan. Diagnosis as an integral part of multi-agent adaptability. Tech Report CS-TR-99-03, UMass, 1999.

[10] I. Muslea, S. Minton, and C. Knobloch. Stalker: Learning expectation rules for simistructured web-based information sources. Papers from the 1998 workshop on ai and information gathering. technical report ws-98-14, University of Southern California, 1998.

[11] H. S. Nwana, D. T. Ndumu, L. C. Lee, and J. C. Collis. ZEUS: A toolkit for building distributed multi-agent systems. (6), 1998.

[12] A. Rao and M. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 312–319, San Francisco, June 1995. AAAI Press.

[13] L. Spector, J. Hendler, and M. P. Evett. Knowledge representation in parka. Technical Report CS-TR-2410, University of Maryland, 1990.

[14] K. Sycara, K. S. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, 11(6):36–46, Dec. 1996.

[15] T. Wagner, A. Garvey, and V. Lesser. Criteria-Directed Task Scheduling. *International Journal of Approximate Reasoning, Special Issue on Scheduling*, 19(1-2):91–118, 1998. A version also available as UMASS CS TR-97-59.

[16] M. Williamson, K. S. Decker, and K. Sycara. Unified information and control flow in hierarchical task networks. In *Proceedings of the AAAI-96 workshop on Theories of Planning, Action, and Control*, 1996.

[17] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, October, 1994.